HybridTSS: A Recursive Scheme Combining Coarse- and Fine- Grained Tuples for Packet Classification

Yuxi Liu[§†*], Yao Xin[†*], Wenjun Li[★†](✉: wenjunli@seas.harvard.edu), Haoyu Song[‡], Ori Rottenstreich[¶], Gaogang Xie[◇△], Weichao Li[†], Yi Wang[§†]

[§]Institute of Future Networks in SUSTech [†]Peng Cheng Laboratory [★]Harvard University [‡]Futurewei
[¶]Technion [◇]CNIC of the Chinese Academy of Sciences [△]University of Chinese Academy of Sciences

ABSTRACT

The popular OpenFlow virtual switch Open vSwitch (OVS) uses a variant of Tuple Space Search (TSS) for packet classification. Although it is easy for rule updates, the lookup performance is poor. By introducing partial trees into TSS, the recently proposed CutTSS improves the lookup performance of TSS. However, it is challenging to replace TSS in OVS for two reasons: (1) the hand-tuned partitioning heuristics are rule-set dependent; (2) the complex and irregular data structures make it difficult to be integrated and maintained in real systems. To address these issues, we propose HybridTSS, a recursive TSS scheme for fast packet classification in OVS, which exploits three novel ideas: (1) the recursive partitioning based on reinforcement learning balances global rule partitions with low training complexity; (2) a hybrid TSS scheme combining coarse-grained and fine-grained tuples suppresses tuple explosion in TSS; (3) a heterogeneous search algorithm consisting of TSS and linear search adapts to characteristics of rules at different scales for fast lookups. Using ClassBench, we show that, while immune from the main drawbacks of CutTSS, HybridTSS retains the update performance of TSS, and achieves almost an order of magnitude higher lookup performance than TSS, making it an ideal packet classification algorithm for OVS.

∗Yuxi Liu and Yao Xin contributed equally to this paper, and they conducted this work in the Peng Cheng Laboratory under the guidance of corresponding authors Wenjun Li and Yi Wang. Yi Wang is also with Heyuan Bay Area Digital Economy Technology Innovation Center. The source code of this paper can be downloaded from the website (http://wenjunli.com/HybridTSS) and the GitHub (https://github.com/wenjunpaper/HybridTSS).

APNet 2022, July 1–2, 2022, Fuzhou, China

ACM ISBN 978-1-4503-9748-3/22/07…$15.00
https://doi.org/10.1145/3542637.3542644

CCS CONCEPTS

• **Networks** → **Packet classification**;

KEYWORDS

SDN, Open vSwitch, packet classification, machine learning

ACM Reference Format:
Yuxi Liu, Yao Xin, Wenjun Li, Haoyu Song, Ori Rottenstreich, Gaogang Xie, Weichao Li, and Yi Wang. 2022. HybridTSS: A Recursive Scheme Combining Coarse- and Fine- Grained Tuples for Packet Classification. In *6th Asia-Pacific Workshop on Networking (APNet 2022), July 1–2, 2022, Fuzhou, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3542637.3542644

1 INTRODUCTION

Backed by software-defined networking (SDN), software virtual switches enable a wide spectrum of non-traditional network functionalities, such as flexible resource partitioning and real-time migration. With the development of network function virtualization (NFV) and cloud service, virtual switches are becoming an important part of virtualized network infrastructures. The popular Open vSwitch (OVS) enforces forwarding policies with OpenFlow table lookups, which is essentially a multi-field packet classification problem [27]. Compared with the packet classification in conventional switches and routers [6], the packet classification in OpenFlow switches faces some new challenges such as higher rule dimensions and faster rule update rates [17], making this open problem more challenging than ever.

As a widely studied bottleneck, packet classification has attracted extensive research attention, and many algorithmic approaches have been proposed in the past two decades [13, 38]. Among them, decision tree and Tuple Space Search (TSS) are two major approaches. The well-researched decision tree based schemes [14, 23, 24, 29, 41, 46] are more promising to achieve high search speed, but the notorious rule duplication problem inflates the memory footprint and hinders the support for fast updates. Moreover, the traditional decision tree based schemes rely on hand-tuned heuristics to construct the tree, making them difficult to adapt to general rule sets with different characteristics and arbitrary number

Table 1: Example rule set with four IPv4 header fields

id	*priority*	src_addr	dest_addr	src_port	dest_port	*action*
R_1	12	228.128.0.0/9	124.0.0.0/7	119:119	0:65535	*action1*
R_2	11	223.0.0.0/9	38.0.0.0/7	20:20	1024:65535	*action2*
R_3	10	175.0.0.0/8	0.0.0.0/1	53:53	0:65535	*action3*
R_4	9	128.0.0.0/1	37.0.0.0/8	53:53	1024:65535	*action4*
R_5	8	0.0.0.0/2	225.0.0.0/8	123:123	0:65535	*action5*
R_6	7	123.0.0.0/8	128.0.0.0/1	0:65535	0:65535	*action6*
R_7	6	0.0.0.0/1	255.0.0.0/8	25:25	0:65535	*action7*
R_8	5	246.0.0.0/7	0.0.0.0/0	0:65535	53:53	*action8*
R_9	4	160.0.0.0/3	252.0.0.0/6	0:65535	0:65535	*action9*
R_{10}	3	0.0.0.0/0	254.0.0.0/7	0:65535	0:65535	*action10*
R_{11}	2	0.0.0.0/1	224.0.0.0/3	0:65535	23:23	*action11*
R_{12}	1	128.0.0.0/1	128.0.0.0/1	0:65535	0:65535	*action12*

of fields. To address this issue, machine learning (ML) has recently been introduced into the packet classification problem to optimize the classifier [16], such as NeuroCuts [25] and NuevoMatch [30, 31]. Nevertheless, real-time rule update is still facing challenges in these decision trees and ML-based methods.

As a result, OVS implements a variant of TSS [28] other than decision trees for its flow table lookups. The primary reason is its good support for rule updates. However, its lookup performance is poor since all the partitioned tuples need to be exhaustively searched. TupleMerge [7] improves TSS by merging tuples with similar characteristics, but its bottom-up method restricts its ability of merging tuples and might cause the performance degradation for large-scale rule sets. The recently proposed CutTSS [22] combines the advantages of both decision tree and TSS to acheive high performance on both table search and rule update. However, it is challenging to replace TSS with CutTSS in OVS for two reasons: (1) the partitioning result based on hand-tuned heuristics highly depends on rule set characteristics; (2) the complex and irregular data structures make its integration and maintenance in real system difficult.

In this paper, we propose a recursive tuple space search scheme, HybridTSS, aided by reinforcement learning (RL) for fast packet classification in OVS. HybridTSS recursively partitions rules into multi-level subsets from top to bottom, and the classifier in each level (or rollout) consists of hybrid TSS structures: coarse-grained tuples, middle-layer hash tables, and fine-grained tuples. A heterogeneous algorithm consisting of TSS and linear search is utilized for search in terminal/bottom subsets. More specifically, in the first stage, reinforcement learning is used to learn optimal field prefixes of the rules to partition a few rule subsets. It enables efficient rule set partitioning without the trouble of rule replication and makes the rule distribution in hash tables more balanced. Instead of implementing a classifier in NuevoMatch, machine learning method in our approach is not coupled to the subsequent classification architecture, so that it can well support incremental rule updates. Partitioned rule subsets correspond to coarse-grained tuple spaces grouped by different prefix combinations from selected fields in current

Table 2: Classifier based on TSS, PSTSS & TupleMerge

PSTSS [28]			**TupleMerge [7]**		
tuple priority	**TSS [36]** tuple	**TSS [36]** rule subset	original tuple	merged tuple	rule subset
12	(9, 7)	R_1, R_2	(9, 7)	(9, 7)	R_1, R_2
10	(8, 1)	R_3, R_6	(8, 1)	(7, 0)	R_3, R_6, R_8
9	(1, 8)	R_4, R_7	(7, 0)		
8	(2, 8)	R_5	(1, 8)	(0, 7)	R_4, R_5, R_7, R_{10}
5	(7, 0)	R_8	(2, 8)		
4	(3, 6)	R_9	(0, 7)		
3	(0, 7)	R_{10}	(3, 6)	(3, 6)	R_9
2	(1, 3)	R_{11}	(1, 3)	(1, 1)	R_{11}, R_{12}
1	(1, 1)	R_{12}	(1, 1)		

level. With the valid bits dertermined by tuple mask as key, hash table is built for each subset in the second stage. If the number of collided rules in an entry exceeds a pre-defined threshold, those rules would be processed in the next level of classifier establishment. This recursive process continues until all the rules are settled. In the third stage, heterogeneous search methods are applied on the terminal subsets with fine-grained TSS and linear search.

Using ClassBench [40], we compare HybridTSS with the other TSS-based approaches. Experiment results show that HybridTSS has similar update performance to the well-known PSTSS [28], while on average achieving 8.63x speed-up on classification speed over PSTSS. HybridTSS also outperforms TupleMerge and CutTSS, two state-of-the-art TSS schemes, with an average of 1.54x and 1.92x speed-up on classification speed respectively, and with an average of 1.44x and 1.45x speed-up on update time respectively. Moreover, HybridTSS has similar data structures as the TSS in OVS, making it easier to be integrated and maintained in real systems.

The rest of the paper is organized as follows. We first summarize the backgroud and related work in Section 2. We then present the technical details of the proposed HybridTSS in Section 3 and provide experimental results in Section 4. Finally, we conclude this work and give our future work in Section 5.

2 BACKGROUD AND RELATED WORK

2.1 The Packet Classification Problem

The purpose of network packet classification is to enable differentiated packet processing based on a classifier that contains a set of predefined rules with priority. Each rule consists of a set of fields in exact value, prefix, or range representations, and the action to be taken when being matched. Table 1 shows a rule set defined over four IPv4 header fields. As an extensively studied problem [38], many algorithmic approaches had been proposed in last two decades, such as decision tree [1, 4, 5, 8, 9, 12, 14, 15, 21, 23, 24, 29, 32–34, 41, 43, 46], decomposition [2, 10, 11, 18, 19, 37, 39, 44, 45], and TSS-based schemes [7, 22, 26, 28, 35, 36, 47]. Next, we briefly describe some related TSS solutions pertaining to our proposed scheme.

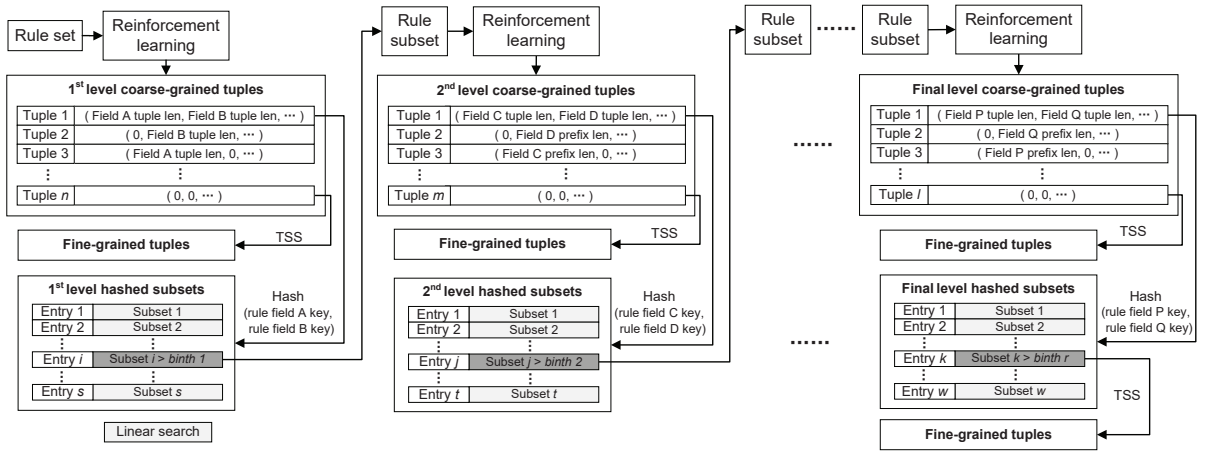


Figure 1: The framework of HybridTSS, the threshold parameter *binths* and MAX recursion level are configurable

2.2 Tuple Space based Solutions

In the original TSS scheme [36], rules are partitioned into a set of tuple spaces (tuple for short) based on prefix lengths, so that the rules mapped to the same tuple can be searched with a hash table. The left part of Table 2 shows an example of TSS constructed based on two IP addresses with the rules in Table 1. As each rule is stored only once, each rule can be inserted/deleted from the hash table in amortized one memory access, resulting in a high update performance. However, in order to find the best matching rule for each packet, all these partitioned hash tables have to be searched, resulting in a low lookup performance. As an improvement, Priority Sorting Tuple Space Search (PSTSS) scheme [28] reduces the average number of table accesses by introducing a pre-computed priority for each tuple and ordering the tuples according to the highest priority of the rules associated with each tuple, as shown in the leftmost column of Table 2. Thus, each search can terminate as soon as a match is found and the matched rule's priority is higher than the priority of the next tuple. Unfortunately, PSTSS has the same worst-case performance as the original TSS.

TupleMerge [7], a recently proposed tuple space scheme, improves upon TSS by relaxing the restrictions on the rules that can be placed in the same tuple. Based on the observation that many rules have similar but non-identical tuples must be placed in separate tables in TSS, TupleMerge allows these rules to be placed into a same table, reducing the number of tables required and leading to faster classification. In the algorithm, the maximum common tuple is found and some bits are ignored for merging similar tuples. The right part of Table 2 shows the example of TupleMerge. However, with more tuples randomly merged, its lookup performance may be affected due to hash collisions. Thus, this bottom-up heuristic approach restricts its ability of grouping rules.

CutTSS [22], a state-of-the-art tuple space approach, adopts a top-down TSS construction method, aided by decision trees for fast lookups. In CutTSS, rules are first divided into several subsets based on a few distinct field, and then decision trees assisted by TSS are built for rule subsets (except the final subset) by cutting on *small fields*. For each lookup or update, the packet or rule will first traverse the trees for the specific terminal operation node. Thus, by exploiting the benefits of decision tree and TSS adaptively, CutTSS not only avoids rule replications, but also supports fast lookups. However, it is challenging to apply CutTSS in OVS due to the aforementioned two reasons.

3 ALGORITHM DESIGN

3.1 Framework of HybridTSS

As shown in Figure 1, the framework of HybridTSS exploits RL to recursively construct a hybrid TSS structure: from coarse-grained to fine-grained tuple spaces. Instead of the bottom-up tuple merging in TupleMerge, HybridTSS adopts a top-down TSS building method. It first uses RL to learn a set of parameters to partition rules without replication into several coarse-grained tuples. Then, the indivisible rules under the current configuration, which are named impartible rules, are allocated to the final tuple labeled (0, 0, ...) in Figure 1. The rule subsets in each tuple other than (0, 0, ...) are hashed into separated tables. For entries with few rules after hashing, a linear search is efficient for rule searching. Otherwise, the rules causing more collisions than the threshold parameter *binth* are assigned for the next-level classifier construction in another RL rollout. In a new rollout, the unselected rule fields are explored to separate the rules into coarse-grained tuples. This process continues until either the maximum recursion level is acheived or no threshold-exceeding rules are left. A heterogeneous search approach is finally applied to different terminals of rule subsets. The impartible rules and the threshold-exceeding rules in the final-level hash tables are searched through TSS, while the rules in the other terminals are linearly searched.

3.2 Top-down TSS Partitioning with RL

The TSS partitioning generates coarse-grained tuples with two goals: (1) the collided rule number in each hashed subset is minimized to facilitate linear search; (2) the number of impartible rules in the last coarse tuple of current level is minimized. The coarse-grained partitioning is based on customized prefix length which is less or equal to original rule prefix length, and this length is called coarse-grained tuple length (tuple length for short) in this paper. The choice of tuple length tends to be random rather than empirical, which involves a huge amount of trial-and-error work.

By observation, our TSS partitioning can be classified as an unsupervised autonomous decision-making problem, in which decision maker needs to take actions in a trial-and-error manner and optimize the policy (or strategy) according to the feedback. This scenario is very suitable for reinforcement learning. Specifically, according to the state of the environment, the agent outputs an action through a strategy function on the environment, then the environment gives the agent a reward and transfers to the next state. Finally, an efficient strategy is found so that the agent can obtain as many rewards from the environment as possible.

Through the mapping from subset partitioning problem to RL, we have established a RL framework to perform coarse-grained tuple partitioning from top to bottom. After partitioning, the learning process has little coupling with the subsequent classification architecture, which means the lookups and rule updates no longer require the participation of RL, unless the entire structure is required to be reconstructed.

In this work, the coarse-grained TSS partition problem is fit to a typical RL system: the state s corresponds to the tuple length, and the environment corresponds to the constructed classifier in the current level which consists of coarse-grained tuples each associated with a hash table or fine-grained tuples. The action a is to incrementally adjust possible tuple lengths of selected fields which are used to divide the rule set and construct the current-level hybrid TSS classifier. For example, assuming the tuple lengths for field combination (A, B) include $(len1_A, len1_B)$, $(len2_A, 0)$, $(0, len2_B)$ in current state, the possible actions include $(len1_A+1, len1_B)$, $(len1_A-1, len1_B)$, $(len1_A, len1_B+1)$, $(len1_A, len1_B-1)$, $(len2_A+1, 0)$, $(len2_A-1, 0)$, $(0, len2_B+1)$, $(0, len2_B-1)$ and unchanged $(len1_A, len1_B)$, $(len2_A, 0)$, $(0, len2_B)$. When the tuple length in a coarse tuple changes, the other tuple values remain unchanged. The keys for hashing are the prefixes defined by tuple lengths in selected fields. Each round of learning would generate a negative reward of $f(c_1 \cdot M + c_2 \cdot N)$ through the statistics of classifier, where M is the number of impartable rules and N is the summation of collided rules in the threshold-exceeding entries in the hash tables, c_1 and c_2 are weight coefficients for above mentioned two numbers, and $f(\cdot)$ is the scaling function. The reward is fed back to agent to train the policy $\pi(a|s)$ in order to maximize the future reward.

The state transition is achieved by the action of incrementally changing bit length for one field at one time step, and the next state s_{t+1} (tuple length) is only related to the current state s_t and the action a, so this model is obviously a Markov Decision Process (MDP). As the environment model is not known, Bellman equation is difficult to be explicitly solved in this case. Thus, in order to be able to learn from the environment, we need to let the agent interact with the environment and get some experiences, then through these experiences to carry out strategy evaluation and strategy iteration, so as to finally get the optimal strategy, which is the basic idea of model-free approach. In this work, Q-learning [42] is utilized to train our partitioning framework, which is a widely adopted temporal-difference learning method. Q-learning is an off-policy algorithm using two control strategies, wherein updated policy is different from the behavior policy. Behavior policy is used to select a new action, and updated policy is used to update the value function which is defined by the following Bellman expectation equation:

$$Q^{\pi}(s_t, a_t) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|s_t, a_t] \tag{1}$$

where $Q^{\pi}(s_t, a_t)$ is the state-action value function (Q value) indicating the cumulative reward on the specified "state-action" on step t under policy $\pi(a_t|s_t)$, and R_i means the reward by the "state-action" on step $i-1$. Discount factor γ defines the importance of future rewards.

In the training, a Q-table is firstly established to record the expected Q value of each state-action combination. The state-action values of the Q-table are initialized to 0 and updated iteratively to provide approximations that continues to improve by following steps:

Step 1: Choose an action in the current state s_t through the ϵ greedy strategy and transit to a new state s_{t+1}, that is, exploring a new action with probability ϵ or choosing an optimal behavior with probability $1-\epsilon$.

Step 2: Partition current-level rule set with the tuple lengths determined by s_{t+1}, and calculate M_{t+1} and N_{t+1}.

Step 3: Update the Q value of previous state by Equation 2:

$$\begin{aligned} Q^{\pi}(s_t, a_t) &= (1-\alpha)Q^{\pi}(s_t, a_t) + \alpha[R_{t+1} + \gamma \max_a Q^{\pi}(s_{t+1}, a)] \\ &= (1-\alpha)Q^{\pi}(s_t, a_t) + \alpha[f(c_1 M_{t+1} + c_2 N_{t+1}) \\ &\quad + \gamma \max_a Q^{\pi}(s_{t+1}, a)] \end{aligned} \tag{2}$$

where α is learning rate, $\max_a Q^{\pi}(s_{t+1}, a)$ is the maximum expected future reward given the new state s_{t+1} and all possible actions at s_{t+1}.

Step 4: Repeat above steps until the state converges to a pre-defined range.

After the above learning for each rollout, we can get the tuple lengths at the terminal state which represent a good choice of field prefix combinations, and accordingly construct a hybrid TSS structure for the rule set in current level.

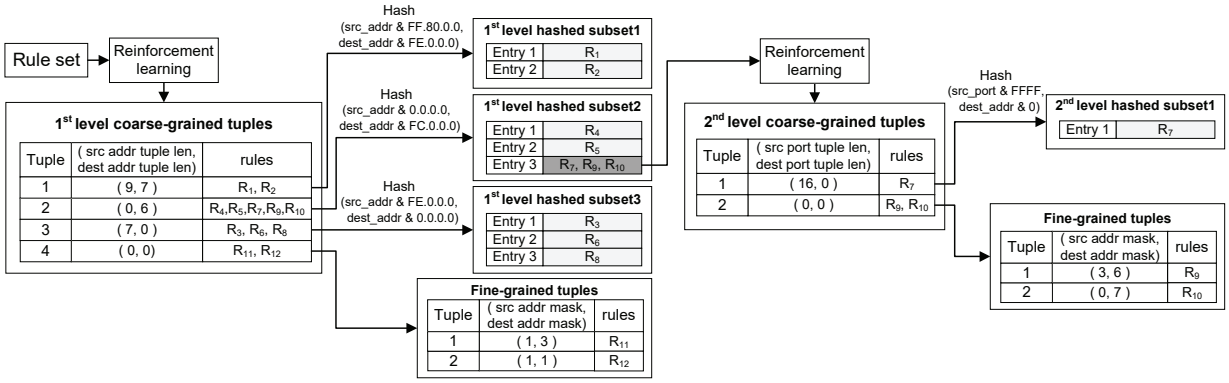


Figure 2: A working example of HybridTSS, with the *binths* = 1 and the MAX recursion level = 2

3.3 Recursive TSS Construction

For a multi-field rule set, each field's distinct properties can be taken advantage of to partition rule set without replication. In each RL training rollout, it is not preferred to select too many fields for combination because it would cause a large number of coarse partitioned subsets as well as long training time. In our experimental evaluations, two fields are selected for one rollout which could be IP addresses or port ranges for IPv4 rules in Table 1. Instead of transforming rule port range to nesting level in the original TSS scheme [36], rule port fields are simply transformed to Longest Common Prefixes (LCP) [3, 20] in this work.

As the recursive classifier construction in HybridTSS is only performed on the aggregated rules in the same hashed entry sharing the common features in the selected fields at the current level, it is possible to further separate them well in the next rollout with newly incorporated field characteristics. Besides, the training time of next rollout will be greatly shortened due to the reduction of rule set size. In fact, one or two levels of recursive rollouts are sufficient for ClassBench rule sets in our experiments, and the majority of rules are generally distributed in the first level.

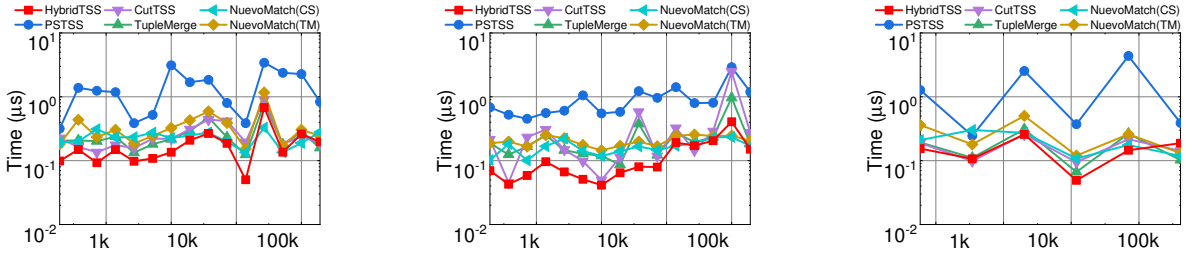
The classifier structure at every level is a hierarchical hybrid tuple space scheme consisting of three stages: coarse-grained tuples, subset hash table, and fine-grained tuples. The coarse tuples partition rules into very few subsets with rules in each set sharing similar characteristics on the selected fields, which facilitates the subsequent hash table establishment because the hash key would exclude wildcard (*) bits to avoid rule replications. The impartible rules account for only a small percentage so that fine-grained tuple search (e.g., PSTSS or TupleMerge) would be efficient on this part. In summary, the overall number of tuples that need to be traversed is greatly reduced, and the distribution of rules in hash tables is balanced through RL, so faster lookup than TSS is achieved. Compared with the decision tree based approach, this scheme has more regular data structures and thus is suitable for OVS implementation.

3.4 A Working Example of HybridTSS

To illustrate HybridTSS more clearly, we walk through an example for 12 rules given in Table 1, as illustrated in Figure 2. Assume that HybridTSS first partitions the rule set into the following four coarse-grained tuple spaces based on source and destination IP addresses: $(9_{src\_addr}, 7_{dst\_addr})$ = $\{R_1, R_2\}$, $(0_{src\_addr}, 6_{dst\_addr})$ = $\{R_4, R_5, R_7, R_9, R_{10}\}$, $(7_{src\_addr}, 0_{dst\_addr})$ = $\{R_3, R_6, R_8\}$, and $(0_{src\_addr}, 0_{dst\_addr})$ = $\{R_{11}, R_{12}\}$. We construct hash tables for the first three tuples correspondingly with different keys of {IP addr value & coarse-grained tuple mask}, while the last subset contains impartible rules which are searched by fine-grained TSS (e.g., PSTSS or TupleMerge). Note that the number of rules in entry 3 of subset 2 exceeds *binth*, so the rules are learned to be partitioned into coarse-grained tuples in second rollout taking into account of the source and destination ports. In the following hash tables, linear search and PSTSS/TupleMerge are applied for the corresponding rules.

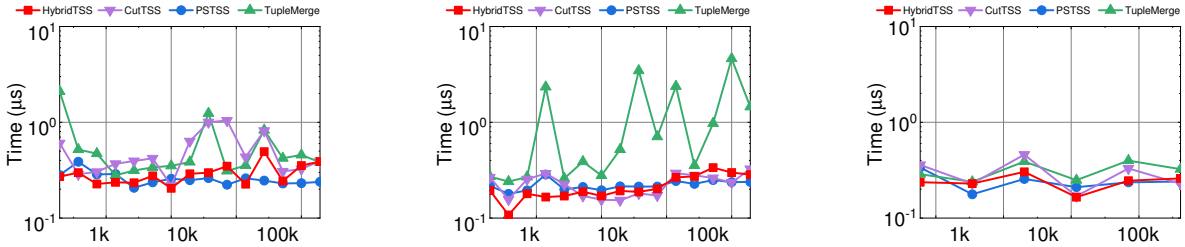
4 EXPERIMENTAL RESULTS

In this section, we evaluate HybridTSS and three representative TSS-based schemes: PSTSS [28], TupleMerge [7], and CutTSS [22]. In addition, HybridTSS is also compared with NuevoMatch [30], which is a state-of-the-art machine learning based packet classifier. Since NuevoMatch can use different algorithms such as CutSplit and TupleMerge to build classifier for the remaining rules after training, we denote them as NuevoMatch(CS) and NuevoMatch(TM), respectively. We evaluate different schemes from the following key aspects: classification performance and update performance. All experiments are run on a PC with Intel Core i7 CPU@3.20GHz and 16G 2400MHz DRAM. The OS is Ubuntu 20.04. The rule sets are generated by ClassBench [40] using default parameters, with rule set size varies from 1k to 100k. There are three types of rule sets: Access Control List (ACL), Firewall (FW) and IP Chain (IPC). For each size, we generate 12 rule sets based on 12 seed parameter files (i.e, 5 ACL, 5 FW, and 2 IPC) in ClassBench.



(a) ACL (b) FW (c) IPC

Figure 3: Average classification time of one packet on three types of rule sets with different sizes



(a) ACL (b) FW (c) IPC

Figure 4: Average update time of one rule on three types of rule sets with different sizes

4.1 Classification Performance

To evaluate classification performance, we generate trace packets ten times the number of rules in each rule set by ClassBench. In order to reduce the influence of CPU jitter, each test is run ten times and the average value is taken as the final result. Figure 3 shows the average classification time of one packet across all rule sets. We can see that the average classification time of HybridTSS is 0.187μs, 0.152μs, 0.119μs in each type of rule set. While PSTSS only reaches 1.452μs, 1.533μs and 0.955μs respectively. HybridTSS achieves 7.76×, 10.09× and 8.03× speed-up compared to PSTSS in terms of classification time. Moreover, HybridTSS achieves 1.92×, 1.54×, 1.82×, and 1.81× average speed-up on all rule sets, when compared to CutTSS, TupleMerge, NuevoMatch(TM), and NuevoMatch(CS) respectively.

4.2 Update Performance

To evaluate update performance, we measure the average time required to conduct a rule update operation by conducting one million rule updates. For each rule set, a rule is randomly selected at each time, and the selection of insertion or deletion is also random. Since NuevoMatch can not support incremental updates well, update performance evaluation is only conducted among HybridTSS, PSTSS, TupleMerge and CutTSS. From Figure 4, we can see HybridTSS is capable of supporting fast rule update with the average time of 0.314 μs, 0.251 μs and 0.219 μs for different types of rule sets, and PSTSS consumes 0.284 μs, 0.255 μs and 0.221 μs in average respectively. Besides, HybridTSS achieves an average of 1.45× and 1.44× speed-up in update time compared with CutTSS and TupleMerge.

5 CONCLUSION AND FUTURE WORK

HybridTSS, a recursive TSS scheme designed for packet classification in OVS, is presented in this paper. It avoids tuple explosion in TSS by recursively partitioning rule sets into multi-layer tuples from top to bottom. At each stage of the framework, we adopt the RL method to build a small number of coarse-grained tuples, which can balance the rule mapping for the subsequent construction of hash tables and fine-grained tuples. Then the rules in each coarse-grained tuple space are hashed into subsets, in which a heterogeneous algorithm is applied for fast searching. Experiment results show that HybridTSS achieves almost an order of magnitude higher lookup performance than TSS. Furthermore, HybridTSS has similar data structures as TSS, making it an ideal alternative packet classification algorithm for OVS.

As our future work (or ongoing work), we will improve HybridTSS from the following aspects: 1) Adopt new ML/RL approaches for globally balanced tuple partitioning, rather than each level in this work; 2) Combine packet classification with flow cache; 3) Integrated to OVS and offload to FPGA.

ACKNOWLEDGMENTS

This work is supported by National Key R&D Program of China (2020YFB1806400), NSFC (61725206, 62102203), Basic Research Enhancement Program of China (2021-JCJQ-JJ-0483), China Postdoctoral Science Foundation (2020TQ0158, 2020M682825), International Postdoctoral Exchange Fellowship Program of China (PC2021037), Key-Area R&D Program of Guangdong Province (2020B0101130003), Guangdong Basic and Applied Basic Research Foundation (2019B1515120031), and the Major Key Project of PCL (PCL2021A02, PCL2021A08).

REFERENCES

[1] Florin Baboescu, Sumeet Singh, and George Varghese. 2003. Packet classification for core routers: Is there an alternative to CAMs?. In *IEEE INFOCOM*.

[2] Florin Baboescu and George Varghese. 2001. Scalable Packet Classification. In *ACM SIGCOMM*.

[3] Yeim-Kuan Chang. 2006. A 2-level TCAM architecture for ranges. *IEEE Trans. Comput.* 55, 12 (2006), 1614–1629.

[4] Yeim-Kuan Chang. 2008. Efficient multidimensional packet classification with fast updates. *IEEE Trans. Comput.* 58, 4 (2008), 463–479.

[5] Yeim-Kuan Chang and Chun-Sheng Hsueh. 2015. Range-enhanced packet classification design on FPGA. *IEEE Transactions on Emerging Topics in Computing* 4, 2 (2015), 214–224.

[6] H. Jonathan. Chao and Bin Liu. 2007. High Performance Switches and Routers. In *John Wiley & Sons, Ltd.*

[7] James Daly and *et al.* 2019. TupleMerge: Fast software packet processing for online packet classification. *IEEE/ACM Transactions on Networking* 27, 4 (2019), 1417–1431.

[8] James Daly and Eric Torng. 2018. ByteCuts: Fast Packet Classification by Interior Bit Extraction. In *IEEE INFOCOM*.

[9] Jeffrey Fong and *et al.* 2012. ParaSplit: A scalable architecture on FPGA for terabit packet classification. In *IEEE Hot Interconnects*.

[10] Filippo Geraci, Marco Pellegrini, Paolo Pisati, and Luigi Rizzo. 2005. Packet classification via improved space decomposition techniques. In *IEEE INFOCOM*.

[11] Pankaj Gupta and Nick McKeown. 1999. Packet classification on multiple fields. In *ACM SIGCOMM*.

[12] Pankaj Gupta and Nick McKeown. 1999. Packet classification using hierarchical intelligent cuttings. In *IEEE Hot Interconnects*.

[13] Pankaj Gupta and Nick McKeown. 2001. Algorithms for packet classification. *IEEE Network* 15, 2 (2001), 24–32.

[14] Peng He, Gaogang Xie, Kavé Salamatian, and Laurent Mathy. 2014. Meta-algorithms for software-based packet classification. In *IEEE ICNP*.

[15] Weirong Jiang and Viktor K Prasanna. 2012. Scalable Packet Classification on FPGA. *IEEE Transactions on Very Large Scale Integration Systems* 20, 9 (2012), 1668–1680.

[16] Marios Evangelos Kanakis, Ramin Khalili, and Lin Wang. 2022. Machine Learning for Computer Systems and Networking: A Survey. *Comput. Surveys* (2022).

[17] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. 2015. What you need to know about SDN flow tables. In *International Conference on Passive and Active Network Measurement*.

[18] TV Lakshman and Dimitrios Stiliadis. 1998. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *ACM SIGCOMM*.

[19] Wenjun Li and *et al.* 2019. Memory-efficient recursive scheme for multi-field packet classification. *IET Communications* 13, 9 (2019), 1319–1325.

[20] Wenjun Li and *et al.* 2019. A power-saving pre-classifier for TCAM-based IP lookup. *Computer Networks* 164 (2019), 106898.

[21] Wenjun Li and *et al.* 2019. TabTree: A TSS-assisted Bit-selecting Tree Scheme for Packet Classification with Balanced Rule Mapping. In *ACM/IEEE ANCS*.

[22] Wenjun Li and *et al.* 2020. Tuple Space Assisted Packet Classification with High Performance on Both Search and Update. *IEEE Journal on Selected Areas in Communications* 38, 7 (2020), 1555–1569.

[23] Wenjun Li and Xianfeng Li. 2013. HybridCuts: A scheme combining decomposition and cutting for packet classification. In *IEEE Hot Interconnects*.

[24] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. 2018. CutSplit: A Decision-Tree Combining Cutting and Splitting for Scalable Packet Classification. In *IEEE INFOCOM*.

[25] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. 2019. Neural packet classification. In *ACM SIGCOMM*.

[26] Hyesook Lim and So Yeon Kim. 2010. Tuple pruning using Bloom filters for packet classification. *IEEE Micro* 30, 3 (2010), 48–59.

[27] Nick McKeown and *et al.* 2008. OpenFlow: Enabling innovation in campus networks. In *ACM SIGCOMM*.

[28] Ben Pfaff and *et al.* 2015. The design and implementation of Open vSwitch. In *USENIX NSDI*.

[29] Yaxuan Qi and *et al.* 2009. Packet classification algorithms: From theory to practice. In *IEEE INFOCOM*.

[30] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. 2020. A Computational Approach to Packet Classification. In *ACM SIGCOMM*.

[31] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. 2022. Scaling Open vSwitch with a Computational Cache. In *USENIX NSDI*.

[32] Ori Rottenstreich and *et al.* 2013. Compressing forwarding tables. In *IEEE INFOCOM*.

[33] Ori Rottenstreich and *et al.* 2020. Cooperative rule caching for SDN switches. In *IEEE CloudNet*.

[34] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. 2003. Packet classification using multidimensional cutting. In *ACM SIGCOMM*.

[35] Haoyu Song, Jonathan Turner, and Sarang Dharmapurikar. 2006. Packet classification using coarse-grained tuple spaces. In *ACM/IEEE ANCS*.

[36] Venkatachary Srinivasan, Subhash Suri, and George Varghese. 1999. Packet Classification using Tuple Space Search. In *ACM SIGCOMM*.

[37] Venkatachary Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. 1998. Fast and Scalable Layer Four Switching. In *ACM SIGCOMM*.

[38] David E Taylor. 2005. Survey and taxonomy of packet classification techniques. *Comput. Surveys* 37, 3 (2005), 238–275.

[39] David E Taylor and Jonathan S Turner. 2005. Scalable packet classification using distributed crossproducing of field labels. In *IEEE INFOCOM*.

[40] David E Taylor and Jonathan S Turner. 2007. ClassBench: A packet classification benchmark. *IEEE/ACM Transactions on Networking* 15, 3 (2007), 499–511.

[41] Balajee Vamanan, Gwendolyn Voskuilen, and TN Vijaykumar. 2010. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *ACM SIGCOMM*.

[42] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3 (1992), 279–292.

[43] Yao Xin and *et al.* 2021. KickTree: A Recursive Algorithmic Scheme for Packet Classification with Bounded Worst-Case Performance. In *ACM/IEEE ANCS*.

[44] Yao Xin and *et al.* 2022. FPGA-based Updatable Packet Classification using TSS-combined Bit-selecting Tree. *IEEE/ACM Transactions on Networking* (2022).

[45] Yang Xu, Zhaobo Liu, Zhuoyuan Zhang, and H. Jonathan Chao. 2014. High-Throughput and Memory-Efficient Multimatch Packet Classification Based on Distributed and Pipelined Hash Tables. *IEEE/ACM Transactions on Networking* 22, 3 (2014), 982–995.

[46] Sorrachai Yingchareonthawornchai, James Daly, Alex X Liu, and Eric Torng. 2016. A sorted partitioning approach to high-speed and fast-update OpenFlow classification. In *IEEE ICNP*.

[47] Xinyi Zhang and *et al.* 2021. Fast Online Packet Classification With Convolutional Neural Network. *IEEE/ACM Transactions on Networking* 29, 6 (2021), 2765–2778.